Problem 4 Editorial: Water Bottles
4+3=7 Points

Problem ID: `bottles`
Rank: `2+3`

Special thanks to Joshc (Joshua Chen) for contributing to this editorial and allowing us to use his Java code.

Overview

Problem statement: https://calico.berkeley.edu/files/calico-fa22/contest/bottles/bottles.pdf

This problem requires an observation about the nature of the optimal ordering of the students. From a contest strategy point of view, it's often useful to try various common orderings, such as sorting sequences in ascending or descending order. It may also be useful to run a brute force solution on small cases in order to spot a pattern.

Main Test Set

Analyzing Optimal Orderings *Solution available in C++*

Let's start by considering small examples—in particular, we will consider $\mathbf{N} = 2$. The wait time of whoever goes second will be the same regardless of the students' order; hence, we should put the student with the smaller capacity bottle first so that the wait time of the first student is minimized.

Now, let's consider the original problem. Consider two adjacent students in any ordering. Observe that if we were to swap the order of these two students, then none of the wait times of the other students would be affected! Using our solution for $\mathbf{N} = 2$, we see that whenever a student with a larger capacity bottle is immediately before a student with a smaller capacity bottle, we should swap their order. This exemplifies the greedy exchange argument.

You might notice that this process is exactly what the bubble sort algorithm does, and indeed, it is optimal to sort the students in ascending order by their bottle's capacity. This makes sense—it is only possible for all adjacent pairs of elements in a list to be in sorted order if the entire list is sorted.

There are multiple ways to sort the students by their bottle capacities. We could store pairs of bottle capacities and their owners, and sort a list of pairs. Alternatively, we could use a custom comparator, or a key function in Python. The time complexity of many common sorting algorithms is $O(\mathbf{N}\log\mathbf{N})$, although slower $O(\mathbf{N}^2)$ sorting algorithms (such as bubble sort) are sufficient to pass the main test set.

We can also use the rearrangement inequality to prove that it is optimal to sort the students by their bottle's capacities. Let $p_1, p_2, \ldots, p_\mathbf{N}$ be some ordering of the bottles' capacities, in the order that their owners are lined up. We see that the total sum of wait times is $(p_1) + (p_1 + p_2) + (p_1 + p_2 + p_3) + \ldots + (p_1 + p_2 + \ldots + p_\mathbf{N}) = \mathbf{N}p_1 + (\mathbf{N} - 1)p_2 + \ldots + p_\mathbf{N}$. The rearrangement inequality tells us that this expression is minimized when $p_1, p_2, \ldots, p_\mathbf{N}$ and $\mathbf{N}, \mathbf{N} - 1, \ldots, 1$ are sorted in opposite ways. Since $\mathbf{N}, \mathbf{N} - 1, \ldots, 1$ is in descending order, we should make $p_1, p_2, \ldots, p_\mathbf{N}$ be in ascending order. Note that we can also reason about the rearrangement inequality intuitively: the x-th student holds up $(\mathbf{N} + 1 - x)$ students after and including themself, so we should ensure that students who hold up more students should take less time.

You could efficiently compute the total wait time as $\mathbf{N}p_1 + (\mathbf{N} - 1)p_2 + \ldots + p_\mathbf{N}$. But $\mathbf{N} \leq 10^3$ in the main test set, so using a nested loop to calculate the sums $p_1, p_1 + p_2, \ldots$ in $O(\mathbf{N}^2)$ is also viable, which was the intended method.

Time: $O(\mathbf{N}^2)$ *Memory:* $O(\mathbf{N})$

Bonus Test Set 1

With the guarantee of distinct bottle capacities removed and a tie breaking requirement put into place, we need to be more mindful of our final ordering.

Filter Students Who Must Move *Solutions available in C++, Java, Python (main | alt)*

We can use the same idea as in the main test set, sorting the students by their bottle's capacities. The only difference is that we would additionally like to minimize the number of students who have to change position.

First, we can generate a sorted list of the bottle capacities. If the x-th bottle's capacity is equal to the x-th element in the list, then the x-th student does not have to move. Otherwise, we add the x-th student to a list of students who must move. Finally, once we have processed all students, we can sort only the students in this new list in ascending order according to their bottle's capacity.

One way to implement this interweaving of fixed and moving students is to iterate through the filtered list and replace students as needed. All of our main model implementations do this.

Another implementation, specific to Python, is to handle this iteration through the builtin iterator type, intuitively allowing us to "extract" on demand the next student that needs to be moved. Aided by iterators and other Python implementation shortcuts like comprehensions, the suite of the `solve` function can be reduced to 5 lines, as demonstrated in the alternate Python model implementation.

An efficient total wait time formula, like the one described above for the main test set, is now needed. An alternative, possibly more intuitive, would be to keep a running prefix sum after adding each new student.

Time: $O(\mathbf{N}\log\mathbf{N})$ *Memory:* $O(\mathbf{N})$